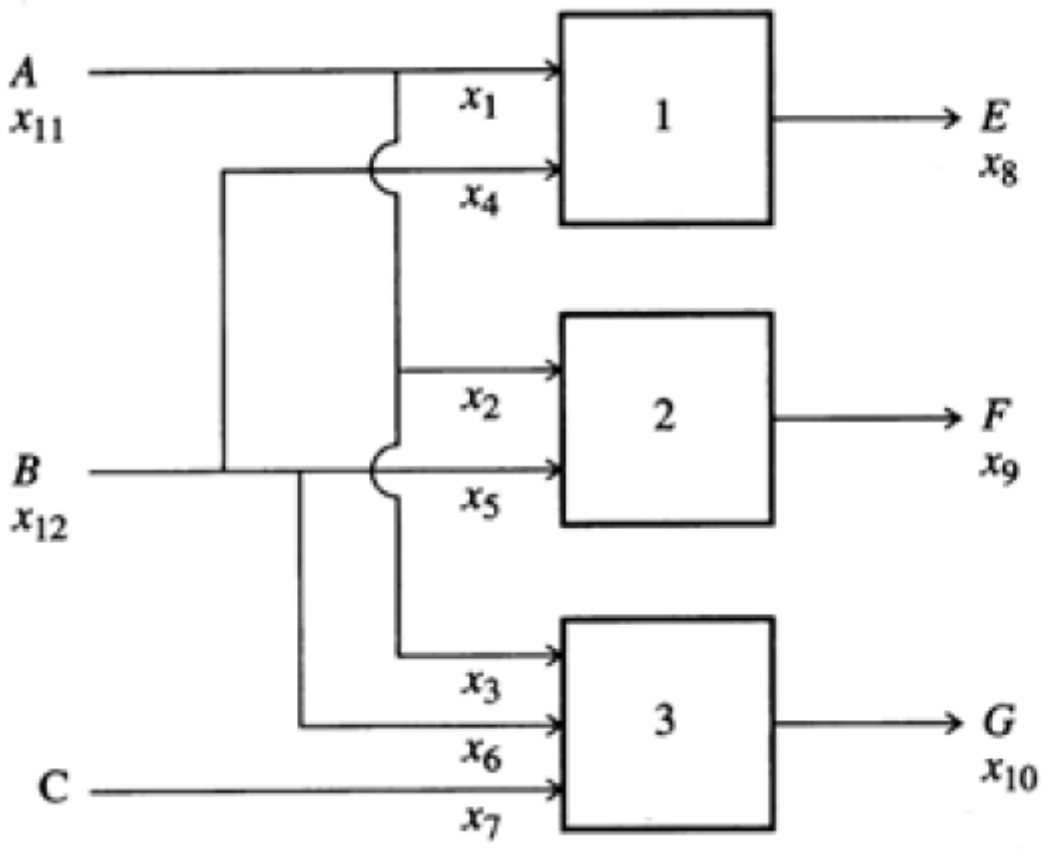
Otimização de Processos (COQ897)

Prof. Argimiro R. Secchi

Terceira Lista de Exercícios - 2020

José Rodrigues Torraca Neto

7) Considerando uma planta projetada para a produção de uma variedade de produtos, conforme ilustrado na figura abaixo:



Processo 1: $A + B \rightarrow E$

Processo 2: $A + B \rightarrow F$

Processo 3: $3A + 2B + C \rightarrow G$

A engenheira Leia ficou encarregada de maximizar o lucro da unidade, limitado à capacidade operacional e sem deixar de atender à demanda do mercado. Os dados de processo econômicos por ela obtidos estão dispostos na tabela abaixo:

Matéria Prima	Oferta (kg/dia)	Custo (R$/kg)	Processo	Produto	Demanda (kg/dia)	Relação Estequiométrica (kg/kg)	Custo de Processa-mento (R$/kg)	Preço de Venda (R$/kg)
A	40.000	1,50	1	E	20.000	2/3A, 1/3B	1,50	4,00
B	30.000	2,00	2	F	15.000	2/3A, 1/3B	0,50	3,30
C	25.000	2,50	3	G	25.000	1/2A, 1/6B, 1/3C	1,00	3,80

(a) Qual foi a formulação do problema de otimização elaborado pela Eng. Leia?

(b) Qual foi a solução obtida?

(c) Qual seria a solução se não houvesse demanda mínima?

Solução:

(a) Primeiro, temos que criar um modelo do processo, que pode ser obtido a partir das equações de balanço de massa.

Definimos x_1, x_2, x_3 como as vazões mássicas de entrada de A para cada processo.

Da mesma forma, sejam x_4, x_5, x_6 as vazões de B e x_7 a vazão de C.

Definindo também as vazões mássicas dos produtos: x_8, x_9, x_{10} , respectivamente para E, F, G.

Sejam x_{11}, x_{12} as quantidades totais de A e B usadas como reagentes (o total de C é o mesmo que x_7).

Assim, temos um total de 12 variáveis.

As restrições de igualdade lineares, do equilíbrio de massa, que representam o processo são:

$$A = x_{11} = x_1 + x_2 + x_3 \tag{1}$$
$$B = x_{12} = x_4 + x_5 + x_6 \tag{2}$$
$$x_1 = \frac{2}{3}x_8 = 0,667x_8 \tag{3}$$
$$x_2 = \frac{2}{3}x_9 = 0,667x_9 \tag{4}$$
$$x_3 = \frac{1}{2}x_{10} = 0,5x_{10} \tag{5}$$
$$x_4 = \frac{1}{3}x_8 = 0,333x_8 \tag{6}$$
$$x_5 = \frac{1}{3}x_9 = 0,333x_9 \tag{7}$$
$$x_6 = \frac{1}{6}x_{10} = 0,167x_{10} \tag{8}$$
$$x_7 = \frac{1}{3}x_{10} = 0,333x_{10} \tag{9}$$

Com 12 variáveis e 9 restrições de igualdade linearmente independentes, temos 3 graus de liberdade que podem ser usados para maximizar o lucro.

A receita total diária em reais da planta é obtida a partir dos preços de venda:

> $Receita (R\$/dia) : 4,00E + 3,30F + 3,80G$

Os custos operacionais diários em reais incluem:

> $Custo\ da\ matéria - prima\ bruta : 1,50A + 2,00B + 2,50C$
> $Custo\ de\ processamento : 1,50E + 0,50F + 1,00G$
> $Custo\ total\ (R\$/dia) : 1,50A + 2,00B + 2,50C + 1,50E + 0,50F + 1,00G$

A função objetivo que representa o lucro diário $f(x)$ é obtida subtraindo-se a receita diária pelo custo total diário:

> $f(x) = 2,5E + 2,80F + 2,80G - 1,50A - 2,00B - 2,50C$
> $f(x) = 2,5x_8 + 2,80x_9 + 2,80x_{10} - 1,50x_{11} - 2,00x_{12} - 2,50x_7$

As seis variáveis na função objetivo do lucro são restringidas pelos balanços de massa:

> $x_{11} = 0,667x_8 + 0,667x_9 + 0,5x_{10}$
> $x_{12} = 0,333x_8 + 0,333x_9 + 0,167x_{10}$
> $x_7 = 0,333x_{10}$

Substituindo essas três últimas equações em $f(x)$, temos:

> $f(x) = 0,8335x_8 + 1,1335x_9 + 0,8835x_{10}$

Como é um problema de maximização, podemos considerar a minimização de $-f(x)$:

$$f(x) = -0,8335x_8 - 1,1335x_9 - 0,8835x_{10} \tag{I}$$

Com as seguintes restrições de desigualdade:

> $0 \leq x_{11} \leq 40000$
> $0 \leq x_{12} \leq 30000$
> $0 \leq x_7 \leq 25000$

Substituindo novamente as 3 equações dos balanços de massa nas inequações acima, temos:

$$0 \leq 0,667x_8 + 0,667x_9 + 0,5x_{10} \leq 40000 \tag{II}$$
$$0 \leq 0,333x_8 + 0,333x_9 + 0,167x_{10} \leq 30000 \tag{III}$$
$$0 \leq 0,333x_{10} \leq 25000 \tag{IV}$$

Finalmente, temos as últimas restrições de desigualdade, na produção de E, F, G; a fim de satisfazer a demanda de mercado:

$$x_8 \geq 20000 \tag{V}$$
$$x_9 \geq 15000 \tag{VI}$$
$$x_{10} \geq 25000 \tag{VII}$$

Resposta: (a) A formulação do problema consiste na minimização da função objetivo (I) com as restrições de desigualdade $(II), (III), (IV), (V), (VI), (VII)$.

(b) O problema de otimização descrito no item anterior compreende uma função objetivo linear e restrições lineares, portanto, a programação linear é a melhor técnica para resolvê-lo.

Aqui, iremos utilizar o método ***Simplex***, por meio do pacote ***Scipy***, por se tratar de um problema com pequenas dimensões e baixo número de restrições.

O problema consiste em minimizar:

> $$min_x f(x) = -0,8335x_8 - 1,1335x_9 - 0,8835x_{10}$$

com as restrições:

> $$0,667x_8 + 0,667x_9 + 0,5x_{10} \leq 40000$$
> $$0,333x_8 + 0,333x_9 + 0,167x_{10} \leq 30000$$
> $$0,333x_{10} \leq 25000$$
> $$x_8 \geq 20000$$
> $$x_9 \geq 15000$$
> $$x_{10} \geq 25000$$

Note que as restrições relativas à x_{10} se resumem aos seguintes limites inferiores e superiores ("***bounds***"):

> $$25000 \leq x_{10} \leq 75075,08$$

O problema também já está na forma ***padrão*** de programação linear, ou seja, suas variáveis já estão restritas a valores positivos.

Então, podemos aplicar o método Simplex diretamente, com o auxílio da função ***linprog*** do módulo ***scipy.optimize***:

OBS: A função linprog () resolve apenas problemas de minimização (não maximização) e não permite restrições de desigualdade com o sinal maior ou igual (≥). Ela também leva os limites (zero a infinito positivo) por padrão.

```
#Coeficientes "c" da função objetivo:

c = [-0.8335, -1.1335, -0.8835]

#Matriz "A" das restrições:

A = [[0.667, 0.667, 0.5], [0.333, 0.333, 0.167]]

#Vetor "b" das restrições:

b = [40000, 30000]

#Limites inferiores e superiores ("bounds") das variáveis:

x8_bounds = (20000, None)
x9_bounds = (15000, None)
x10_bounds = (25000, 75075.8)

from scipy.optimize import linprog

opt = linprog(c, A_ub=A, b_ub=b, bounds=[x8_bounds, x9_bounds, x10_bounds], method='revised simplex')

print(opt)

          con: array([], dtype=float64)
          fun: -63101.885
      message: 'Optimization terminated successfully.'
          nit: 2
        slack: array([    0.  , 12782.23])
       status: 0
      success: True
            x: array([20000., 15000., 33310.])
```

A função .linprog () retorna uma estrutura de dados com os seguintes atributos:

- con: são os resíduos das restrições de igualdade.
- fun: é o valor da função objetivo no ponto ótimo (se encontrado).

- message é o status da solução.
- nit: é o número de iterações necessárias para concluir a otimização.
- slack: são os valores das variáveis de folga, ou seja, as diferenças entre os valores dos lados esquerdo e direito das restrições.
- status: é um número inteiro entre 0 e 4 que mostra o status da solução, como 0 para quando a solução ótima for encontrada.
- success: é um booleano que mostra se a solução ótima foi encontrada.
- x: é uma matriz NumPy contendo os valores ótimos das variáveis de decisão.

```
#Podemos acessar esses valores separadamente:

print(-opt.fun)     #Aqui já invertemos o sinal da função objetivo (maximização)

print(opt.success)

print(opt.x)
```

```
63101.885
True
[20000. 15000. 33310.]
```

Resposta: (b) A distribuição ótima de produção é: $20000\ kg/dia$ de E, $15000\ kg/dia$ de F e $33310\ kg/dia$ de G. O lucro máximo é de $63101,885\ R\$/dia$.

(c) O problema é similar ao item anterior, porém sem os limites inferiores ("***inferior bounds***") para as variáveis:

OBS: Mesmo que não existam limites para a variável x_i, nós podemos especificar explicitamente os limites $0 \leq x_i \leq \infty$ ("None") para todas as variáveis. Como observado anteriormente, o padrão do método é que todas as variáveis sejam positivas.

```
#Coeficientes "c" da função objetivo:

c = [-0.8335, -1.1335, -0.8835]

#Matriz "A" das restrições:

A = [[0.667, 0.667, 0.5], [0.333, 0.333, 0.167]]

#Vetor "b" das restrições:

b = [40000, 30000]

#Limites inferiores e superiores ("bounds") das variáveis:

x8_bounds = (0, None)
x9_bounds = (0, None)
x10_bounds = (0, 75075.8)

from scipy.optimize import linprog

opt = linprog(c, A_ub=A, b_ub=b, bounds=[x8_bounds, x9_bounds, x10_bounds], method='revised simplex')

print(opt)
```

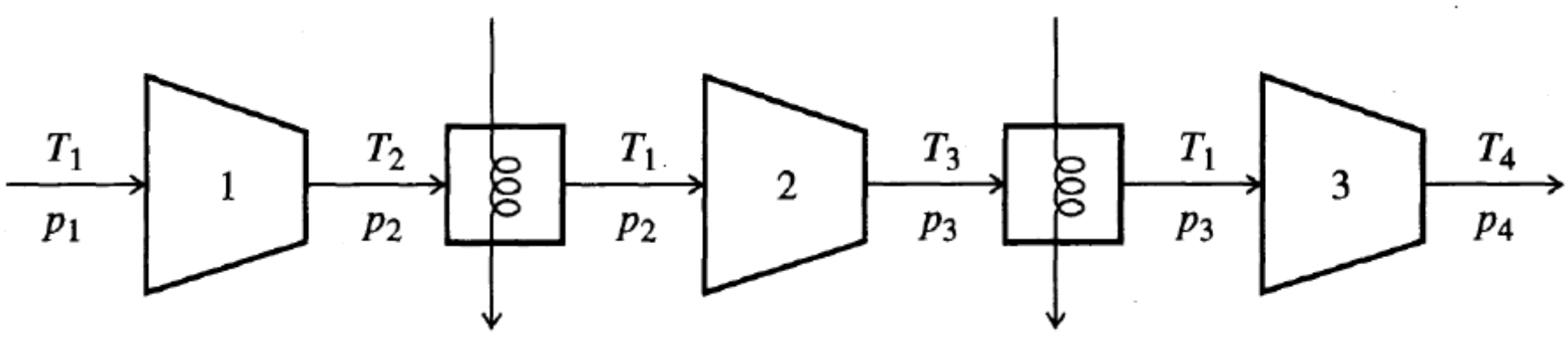
```
          con: array([], dtype=float64)
          fun: -70513.56277826086
      message: 'Optimization terminated successfully.'
          nit: 2
        slack: array([    0.        , 16233.13705217])
       status: 0
      success: True
            x: array([    0.        ,  3691.30434783, 75075.8       ])


#Podemos acessar esses valores separadamente:

print(-opt.fun)

print(opt.success)

print(opt.x)
```

```
70513.56277826086
True
[    0.          3691.30434783 75075.8       ]
```

Resposta: (c) A distribuição ótima de produção seria: $0\ kg/dia$ de E; $3691,3\ kg/dia$ de F e $75075,8\ kg/dia$ de G. O lucro ótimo seria de $70513,6\ R\$/dia$.

8) Determine o trabalho mínimo do sistema de compressão abaixo, considerando compressão adiabática de gás ideal:

$$W = \frac{kRT_1}{k-1}\left[\left(\frac{p_2}{p_1}\right)^{\frac{k-1}{k}} + \left(\frac{p_3}{p_2}\right)^{\frac{k-1}{k}} + \left(\frac{p_4}{p_3}\right)^{\frac{k-1}{k}} - 3\right]$$



Dados $k = 1,4,\ p_1 = 100\ kPa,\ p_4 = 1000\ kPa$ e $T_1 = 300K$.

(a) Esboce a função objetivo e suas curvas de níveis em função de p_2 e p_3.

(b) Utilize os métodos de Rosenbrock, Powell, steepest descent, gradiente conjugado, BFGS e DFP para obter a solução ótima e compare os resultados obtidos.

Solução:

(a) Queremos determinar as pressões ótimas p_2 e p_3, para minimizar a função objetivo W mantendo p_1 e p_4 fixos.

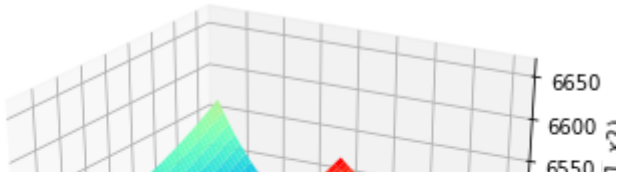
Lembrando que $R = 8,314462\ \ L \cdot kPa \cdot K^{-1} \cdot mol^{-1}$

Substituindo as constantes na expressão de W, e chamando W de $f(x)$; p_2 e p_3 de x_1 e x_2 respectivamente, temos o seguinte problema de otimização :

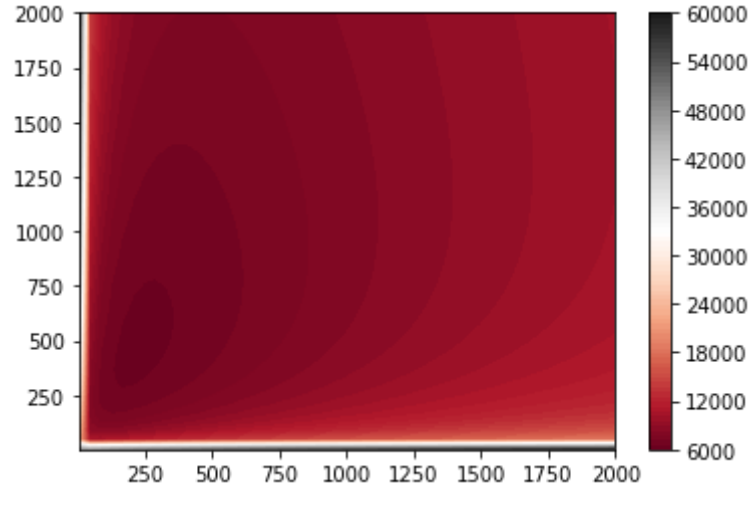
$$\min_{x_1,x_2} f(x_1, x_2) = 8730,1851\left[\left(\frac{x_1}{100}\right)^{0,2857} + \left(\frac{x_2}{x_1}\right)^{0,2857} + \left(\frac{1000}{x_2}\right)^{0,2857} - 3\right]$$

```
import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.optimize import minimize
```

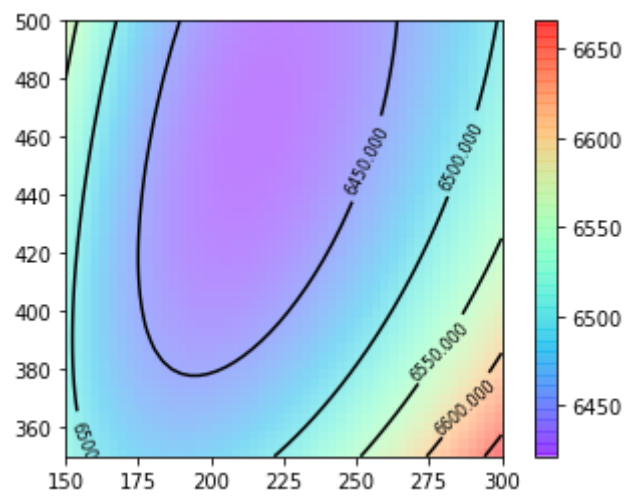
```
#Verificando a forma da superfície descrita pela função objetivo:

from matplotlib import cm

x1 = np.linspace(150, 300, 50)
x2 = np.linspace(350, 500, 50)
X, Y = np.meshgrid(x1, x2)

Z = 8730.1851*((X/100)**0.2857 + (Y/X)**0.2857 + (1000/Y)**0.2857 - 3)

fig, ax = plt.subplots(subplot_kw={'projection': '3d'})
ax.plot_surface(X, Y, Z, cmap=cm.rainbow)

ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$f(x1,x2)$');
```




```
#Plotando superfície de contorno (com colorbar):

plt.contourf(X, Y, Z, 100, cmap='RdGy')
plt.colorbar();
plt.show()
```

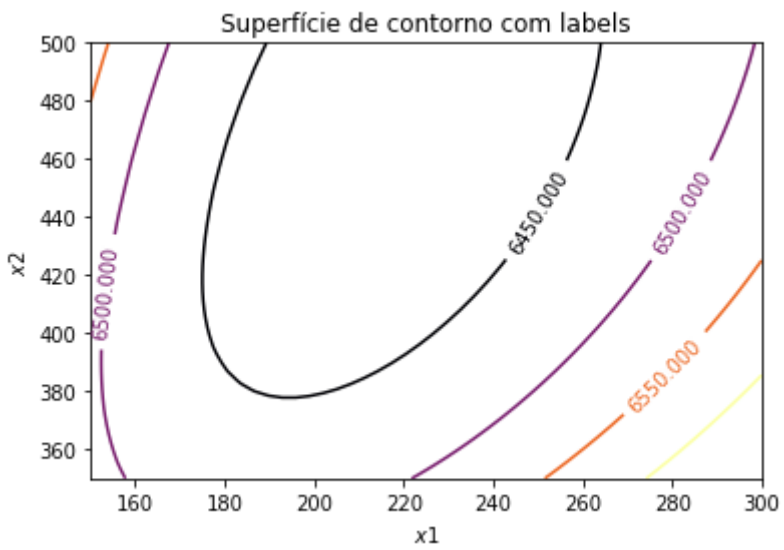



```
#Plotando superfície de contorno (com colorbar) - com labels:

contours = plt.contour(X, Y, Z, 5, colors='black')
plt.clabel(contours, inline=True, fontsize=8)

plt.imshow(Z, extent=[150, 300, 350, 500], origin='lower',
           cmap='rainbow', alpha=0.5)
plt.colorbar();
```




```
#Plot superfície de contorno (padrão) - com labels:

fig, ax = plt.subplots()
CS = ax.contour(X, Y, Z, [6450,6500,6550,6600], cmap='inferno')
ax.clabel(CS, inline=1, fontsize=10)
ax.set_title('Superfície de contorno com labels')
ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
```

```
Text(0, 0.5, '$x2$')
```



(b) Podemos dividir os métodos mencionados no problema em 2 tipos:

- Métodos diretos: não fazem uso da derivada de f(x) **["Rosenbrock" e "Powell"]**.
- Métodos indiretos: fazem uso da derivada de f(x) **["steepest descent", "gradiente conjugado"]**; e em alguns casos também da Hessiana de f(x) **["BFGS" e "DFP"]**.

Os métodos diretos também são conhecidos como DFO ("Derivative Free Optimization"), e os indiretos também são conhecidos como analíticos.

Todos os métodos pedidos serão implementados em algoritmos na linguagem ***Python***, e os indiretos também com o auxílio de um pacote gerenciador de "tabelas/planilhas" conhecido como "***pandas***".

Para podermos fazer comparações coerentes entre os métodos, todos devem possuir o mesmo método de busca de direção ("***linesearch***"), que aqui será considerado como o método da seção áurea ("***golden search***").

Além disso, todos os métodos possuirão um critério de parada, com o mesmo valor para a tolerância (épsilon) de $1 \cdot 10^{-9}$. Não teremos um critério de parada de número máximo de iterações, porque queremos comparar o número de iterações de cada método; mas esse critério também pode ser facilmente imposto com uma ou duas linhas de código se necessário.

Todos os métodos também partirão do ponto inicial arbitrário: $x = (100, 100)$**.** Além disso, os método de Powell e BFGS também serão executados utilizando o pacote Scipy, a fim de comparar os métodos diretos e indiretos respectivamente, com os nossos métodos já implementados.

Finalmente, as informações mais relevantes de cada método serão dispostas em um dataframe, em conjunto com uma visualização gráfica do "caminho" do método sobre as curvas de níveis da função objetivo.

1º Método de otimização direto - ***Rosenbrock***:

```
# Rosenbrock é um método de busca multivariável parecido com a fase de exploração
# do método de Hooke  &  Jeeves.

# Entretanto,  ao  invés  de  continuamente explorar  nas  direções  dos  eixos  coordenados,
# são  construídas  novas  direções ortogonais,  usando  o  procedimento  de  Gram-Schmidt,
# baseadas  nos  tamanhos  dos passos das direções bem sucedidas.

import numpy as np
import pandas as pd
from numpy import linalg as LA
import matplotlib.pyplot as plt
import math
import sympy
from sympy.utilities.lambdify import lambdify

#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══

# Primeiro, vamos simplificar a expressão da função objetivo em termos simbólicos,
# a ajuda do pacote Sympy.

# Extraindo a função objetivo f, em termos simbólicos (.sympy.lambdify):

def f_x(f_expression, values):
    f = lambdify((v[0],v[1]), f_expression)
    return f(values[0],values[1])

v = sympy.Matrix(sympy.symbols('x[0] x[1]'))

# Criando a função objetivo do exercício (8):

f_sympy8 = 8730.1851*((v[0]/100)**0.2857 + (v[1]/v[0])**0.2857 + (1000/v[1])**0.2857 - 3)

# Alocando-a como "f":

f = f_sympy8

# Extraindo a função:

f = f_x(f, v)
print('A função objetivo é: ',f)
print()

#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══
```

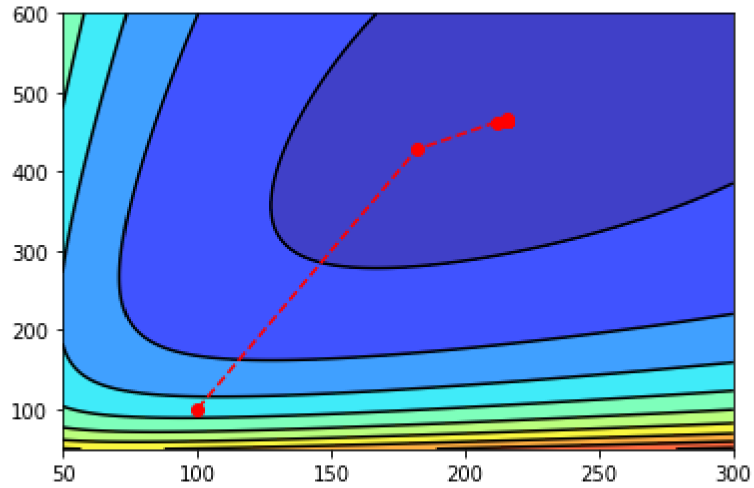
```
     A função objetivo é:  2342.1971697181*x[0]**0.2857 + 8730.1851*(x[1]/x[0])**0.2857 + 62823.6915040394*(1/x[1])**0.2857 - 26190.
```

```
import numpy as np
```

```
import math
import matplotlib.pylab as pl

# Inicializando o contador externamente

count = 0

# Definindo uma função que dá a norma de um vetor (eleva ao quadrado e tira a raiz de cada elemento i)

def norm(s1):                       # Na realidade já existe o método np.linalg(LA).norm que é idêntico
    normas = 0
    for i in range(len(s1)):
        normas += s1[i]**2
    return math.sqrt(normas)

# Definindo um contador que entrará junto com a função fun(x), que dirá quantas vezes fun(x) foi avaliada

def incCount():
    global count
    count = count+1

# Definindo uma função que retorna um vetor "b" de coordenadas perpendicular a um vetor "a" de entrada
# (são arrays 2x2)

def perpendicular(a):
    b = np.empty_like(a)
    b[0] = -a[1]
    b[1] = a[0]
    return b

# Definindo a função fun(x)

def fun(x):
    incCount()
    return 8730.1851*((x[0]/100)**0.2857 + (x[1]/x[0])**0.2857 + (1000/x[1])**0.2857 - 3)

#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══

# Aqui, vamos criar uma função plot para visualizar o caminho que o método realiza sobre as curvas de níveis da função objetivo.

def plot(points):

    n = 256
    x = np.linspace(50, 300, n)
    y = np.linspace(50, 600, n)
    X, Y = np.meshgrid(x, y)

    xs = []
    ys = []

    pl.contourf(X, Y, fun([X, Y]), 8, alpha=.75, cmap='jet')
    C = pl.contour(X, Y, fun([X, Y]), 8, colors='black', linewidth=.5)

    for i in range(len(points)):
        xs.append(points[i][0])
        ys.append(points[i][1])

    pl.plot(xs, ys, marker='o', linestyle='--', color='r', label='Square')

#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══

# Definindo o método de Rosenbrock ("rosen")


def rosen(x0, lmb, epsilon1, epsilon2, alpha, beta):

# x0 : ponto inicial, definido como uma array de entrada [x, y]

# lmb : delta x, array que pode ser definida como [0.1, 0.1]

# epsilon1, epsilon2 : erros de estimação ~ tol

# epsilon1: ||x^2 - x^1|| / ||x^2||

# epsilon2: |f(x^2) - f(x^1)| / |x^2|

# alpha, beta - coeficientes para aumentar/diminuir cada passo, respectivamente.
# Podem ser definidos como (3.0, -0.5)
```

```
# success = array para salvar respostas YES/NO

    start = x0
    s1 = [1, 0]                   # vetor s1 (ou direção d1), inicialmente [1,0]
    s2 = [0, 1]                   # vetor s2 (ou direção d2), inicialmente [0,1]
    N = 0                         # N: nº de iterações
    success1 = []
    success2 = []
    points = []
    points.append(x0)
    iterations = 0

    temp = []

    while True:
        temp = x0
        x1 = [x0[0] + lmb[0]*s1[0], x0[1] + lmb[0]*s1[1]]

        if fun(x1) < fun(x0):
            success1.append("Y")
            lmb[0] *= alpha
            x0 = x1[:]

        elif fun(x1) > fun(x0):
            success1.append("N")
            lmb[0] *= beta

x2 = [x0[0] + lmb[1]*s2[0], x0[1] + lmb[1]*s2[1]]


if fun(x2) < fun(x0):
    success2.append("Y")
    lmb[1] *= alpha
    x0 = x2[:]
elif fun(x2) > fun(x0):
    success2.append("N")
    lmb[1] *= beta

if success1[len(success1)-1] == "N" and success2[len(success2)-1] == "N" and "Y" in success1 and "Y" in success2:
    print("Our stop point: [%8.5f, %8.5f]" % (x0[0], x0[1]))
    s1 = [x0[0] - start[0], x0[1] - start[1]]
    print("S1" + str(s1))
    norma = norm(s1)
    s1[0] /= norma
    s1[1] /= norma
    s2 = perpendicular(s1)
    print("F(X1) = %8.5f" % fun(x0))

    error_point = norm([x0[0] - start[0], x0[1] - start[1]])/norm(x0)
    error_func = abs(fun(x0) - fun(start))/abs(fun(start))

    print("||x^2 - x^1|| / ||x^2|| = %8.5f" % error_point)
    print("|f(x^2) - f(x^1)| / |x^2| = %8.5f" % error_func)

            if error_point < epsilon1 or error_func < epsilon2 and N>1:
                print("DONE")
                print("N = %3d" % N)
                break

            points.append(x0)
            start = x0[:]
            success1 = []
            success2 = []
            N+=1

    plot(points)
    print("Nº de avaliações da função objetivo = " + str(count))
    print("Ponto ótimo = " + str(x0))
    print("Função objetivo no ótimo = %8.5f" % fun(x0))


def main():
    x0 = [100, 100]
    rosen(x0, [0.1, 0.1], 1e-9, 1e-9, 3.0, -0.5)


if __name__ == '__main__':
   main()
```

```
Our stop point: [182.00000, 428.00000]
S1[82.00000000000003, 328.0]
F(X1) = 6440.28807
||x^2 - x^1|| / ||x^2|| =  0.72695
|f(x^2) - f(x^1)| / |x^2| =  0.20731
Our stop point: [211.83643, 462.80917]
S1[29.836429422438414, 34.809167659511445]
F(X1) = 6421.70698
||x^2 - x^1|| / ||x^2|| =  0.09007
|f(x^2) - f(x^1)| / |x^2| =  0.00289
Our stop point: [215.69346, 465.83225]
S1[3.857035191553962, 3.023081636623374]
F(X1) = 6421.49920
||x^2 - x^1|| / ||x^2|| =  0.00955
|f(x^2) - f(x^1)| / |x^2| =  0.00003
Our stop point: [215.85159, 464.75415]
S1[0.15812131894233517, -1.0780999018795683]
F(X1) = 6421.49271
||x^2 - x^1|| / ||x^2|| =  0.00213
|f(x^2) - f(x^1)| / |x^2| =  0.00000
Our stop point: [215.57278, 464.55958]
S1[-0.2788100263717297, -0.19456527020190606]
F(X1) = 6421.49074
||x^2 - x^1|| / ||x^2|| =  0.00066
|f(x^2) - f(x^1)| / |x^2| =  0.00000
Our stop point: [215.54452, 464.40081]
S1[-0.028256550231560595, -0.15877490130117167]
F(X1) = 6421.49044
||x^2 - x^1|| / ||x^2|| =  0.00031
|f(x^2) - f(x^1)| / |x^2| =  0.00000
Our stop point: [215.44248, 464.07149]
S1[-0.10204274836712557, -0.32931977882122965]
F(X1) = 6421.49025
||x^2 - x^1|| / ||x^2|| =  0.00067
|f(x^2) - f(x^1)| / |x^2| =  0.00000
Our stop point: [215.44032, 464.17289]
S1[-0.0021574167972460145, 0.10139858947025004]
F(X1) = 6421.49023
||x^2 - x^1|| / ||x^2|| =  0.00020
|f(x^2) - f(x^1)| / |x^2| =  0.00000
Our stop point: [215.44672, 464.15498]
S1[0.006395954007729188, -0.017908671221618988]
F(X1) = 6421.49022
||x^2 - x^1|| / ||x^2|| =  0.00004
|f(x^2) - f(x^1)| / |x^2| =  0.00000
DONE
N =   8
Nº de avaliações da função objetivo = 372
Ponto ótimo = [215.4467151451748, 464.1549793621796]
Função objetivo no ótimo = 6421.49022
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:53: UserWarning: The following kwargs were not used by contour: 'l
```



2º Método de otimização direto - ***Powell***:

```
# Powell é  um  método  de  busca  multivariável  ao  longo  de  direções  conjugadas

# Duas direções d1  e d2 são ditas conjugadas entre si, com respeito a uma matriz H, se:
# d1.T  H d2  = 0

# A  ortogonalidade  é  um  caso  particular  de  conjugância,  onde  H  =  I.

# O método de Powell também permite que a função seja não-diferenciável, já que não usa suas derivadas.

# No entanto, ele usa uma informação adicional em relação ao Rosenbrock, que é o alfa (tamanho do passo na direção d)

# obtido por um método de otimização monovariável, no caso aqui sendo o método da seção áurea ("golden search").

#═════════╬═════════╬═════════╬═════════╬═════════╬═════════╬═════════╬═════════╬═════════╬═════════╬═════════╬═════════╬═════════╬══
```

ADICIONAL: Método de otimização direto - ***Powell*** por meio do Scipy.optimize.minimize:

```
import numpy as np
from scipy.optimize import minimize

x0 = np.array([100, 100])

# Definindo a função objetivo f(x):

def f(x):
    return 2342.1971697181*x[0]**0.2857 + 8730.1851*(x[1]/x[0])**0.2857 + 62823.6915040394*(1/x[1])**0.2857 - 26190.5553

res = minimize(f, x0, method='powell', options={'ftol': 1e-9, 'disp': True})

res.x
```

```
     Optimization terminated successfully.
              Current function value: 6421.490224
              Iterations: 5
              Function evaluations: 175
     /usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:9: RuntimeWarning: invalid value encountered in double_scalars
       if __name__ == '__main__':
     array([215.44346647, 464.158889  ])
```

1º Método de otimização indireto - ***Gradiente (Steepest Descent)***:

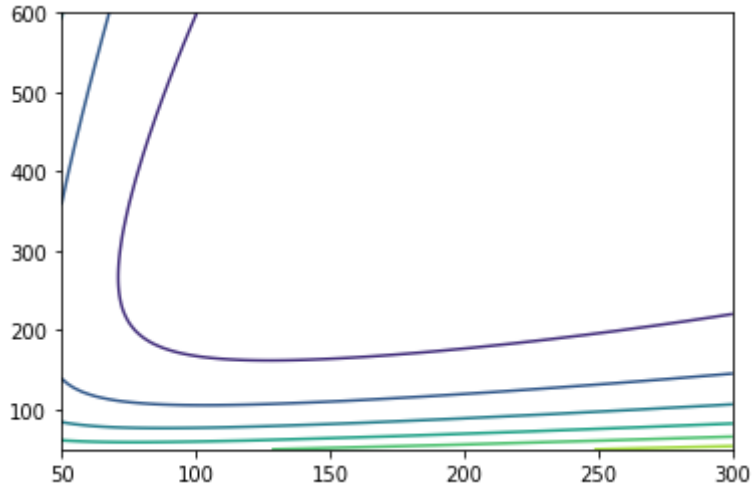
```
# Primeiro, vamos implementar o método steepest descent em python (completo com visualização gráfica):

# Lembrando que o método steepest descent é um método indireto, ou seja, ele faz uso da derivada de f (df_x).

# Nesse bloco de código, vamos utilizar métodos simbólicos para o cálculo dos gradientes (sympy).

# Além disso, outros métodos como Newton fazem uso da Hessiana explicitamente, então é conveniente calcular aqui também.

import numpy as np
import pandas as pd
from numpy import linalg as LA
import matplotlib.pyplot as plt
import math
import sympy
from sympy.utilities.lambdify import lambdify

# Extraindo a função objetivo f, em termos simbólicos (.sympy.lambdify):

def f_x(f_expression, values):
    f = lambdify((v[0],v[1]), f_expression)
    return f(values[0],values[1])

# Extraindo os gradientes de f:

def df_x(f_expression, values):
    df1_sympy = np.array([sympy.diff(f_expression, i) for i in v])       #derivadas de 1ª ordem
    dfx_0 = lambdify((v[0],v[1]), df1_sympy[0])                          #derivada em relação à x_0
    dfx_1 = lambdify((v[0],v[1]), df1_sympy[1])                          #derivada em relação à  x_1
    evx_0 = dfx_0(values[0], values[1])                                  #calculando o gradiente nos valores dados
    evx_1 = dfx_1(values[0], values[1])
    return(np.array([evx_0,evx_1]))

# Extraindo a Hessiana:

def hessian(f_expression):
    df1_sympy = np.array([sympy.diff(f_expression, i) for i in v])             #derivadas de 1ª ordem
    hessian = np.array([sympy.diff(df1_sympy, i) for i in v])                  #Hessiana
    return(hessian)

#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══

# Aqui, vamos criar 2 funções para visualizar o caminho que cada método realiza sobre as curvas de níveis da função objetivo.

# Função para criar um plot de curvas de níveis:

def contour(sympy_function):
    x = np.linspace(50, 300, 100)
    y = np.linspace(50, 600, 100)
    x, y = np.meshgrid(x, y)
    func = f_x(sympy_function, np.array([x,y]))
    return plt.contour(x, y, func)

# Função para plotar com o caminho do algoritmo:
```

```
def contour_travel(x_array, sympy_function):
    x = np.linspace(50, 300, 100)                          #x-axis
    y = np.linspace(50, 600, 100)                          #y-axis
    x, y = np.meshgrid(x, y)                                #criando o grid x & y
    func = f_x(sympy_function, np.array([x,y]))
    plt.contour(x, y, func)
    plot = plt.plot(x_array[:,0],x_array[:,1],'x-')
    return (plot)

v = sympy.Matrix(sympy.symbols('x[0] x[1]'))
```

```
#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══
```

```
# Criando a função objetivo do exercício (8):

f_sympy8 = 8730.1851*((v[0]/100)**0.2857 + (v[1]/v[0])**0.2857 + (1000/v[1])**0.2857 - 3)

# Alocando-a como "f":

f = f_sympy8

# Extraindo a função:

f = f_x(f, v)
print('A função objetivo é: ',f)
print()

# Encontrando o gradiente da função:

df = df_x(f, v)
print('O gradiente da função é: ',df)
print()

# Encontrando a Hessiana da função:             # A etapa da Hessiana é opcional,
                                                # porque não vamos usá-la explicitamente nos métodos aqui considerados,
hess_f = hessian(f)                             # que fazem aproximações de H.
print('A Hessiana da função é: ',hess_f)
print('')

# Plotando as curvas de níveis:

con = contour(f)
```

```
     A função objetivo é:  2342.1971697181*x[0]**0.2857 + 8730.1851*(x[1]/x[0])**0.2857 + 62823.6915040394*(1/x[1])**0.2857 - 26190.

     O gradiente da função é:  [669.165731388461*x[0]**(-0.7143) - 2494.21388307*(x[1]/x[0])**0.2857/x[0]
      2494.21388307*(x[1]/x[0])**0.2857/x[1] - 17948.7286627041*(1/x[1])**0.2857/x[1]]

     A Hessiana da função é:  [Derivative([669.165731388461*x[0]**(-0.7143) - 2494.21388307*(x[1]/x[0])**0.2857/x[0], 2494.21388307*
      Derivative([669.165731388461*x[0]**(-0.7143) - 2494.21388307*(x[1]/x[0])**0.2857/x[0], 2494.21388307*(x[1]/x[0])**0.2857/x[1]
```




```
# Inicializando o contador externamente

count = 0

# Definindo o contador que entrará junto com a função fun(x), que dirá quantas vezes fun(x) foi avaliada

def incCount():
    global count
    count = count+1

# Definindo a função objetivo fo(x1,x2):

def fo(x1,x2):
```

```
def fo(x1,x2):
    incCount()
    return 2342.1971697181*x1**0.2857 + 8730.1851*(x2/x1)**0.2857 + 62823.6915040394*(1/x2)**0.2857 - 26190.5553


# Definindo o gradiente da função:

def gradf(x1,x2):
    return np.matrix([[669.165731388461*x1**(-0.7143) - 2494.21388307*(x2/x1)**0.2857/x1],
                      [2494.21388307*(x2/x1)**0.2857/x2 - 17948.7286627041*(1/x2)**0.2857/x2]])

# Visualização gráfica da função para certificar que está correta:

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

fig = plt.figure()
ax = fig.gca(projection='3d')

X = np.linspace(150, 300, 50)
Y = np.linspace(350, 500, 50)
X, Y = np.meshgrid(X, Y)
Z = fo(X,Y)

surf = ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$f(x1,x2)$');

plt.show()

#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══

# Definindo o método da seção áurea ("golden search") customizado para minimizar o "alfa" (passo). "(ak, bk)" é o intervalo.

# "length_uncertainty" é a incerteza admitida para a solução. "Y" é o ponto inicial e "d" a direção de busca:

def golden_search(ak, bk,length_uncertainty, Y, d):

    k=1
    termination=True

    # Definindo as direções X1 e X2 e o valor do ponto inicial:

    x1_in=float(Y[0])
    x2_in=float(Y[1])
    d1=float(d[0])
    d2=float(d[1])

    # Inicialização. Calculando alfa, u e os valores das funções:

    gama=((-1+math.sqrt(5))/2)                      #gama é o "nº áureo" (0,618)
    alfak=gama*ak+(1-gama)*bk
    uk=(1-gama)*ak+gama*bk

    # Calculando os valores das funções usando o ponto e a direção dados, e o alfa e u calculados:

    falfak=fo(x1_in+alfak*d1, x2_in+alfak*d2)
    fuk=fo(x1_in+uk*d1,x2_in+uk*d2 )

    while termination:

        # Definindo novos valores para o intervalo:

        if falfak<fuk:
            ak=ak
            bk=uk
            uk=alfak
            alfak=gama*ak+(1-gama)*bk
            fuk=falfak
            falfak=fo(x1_in+alfak*d1, x2_in+alfak*d2)

        else:
            ak=alfak
            bk=bk
            alfak=uk
            uk=(1-gama)*ak+gama*bk
            falfak=fuk
            fuk=fo(x1_in+uk*d1,x2_in+uk*d2 )
```

```
        # Checando o critério de parada:

        if (bk-ak)<length_uncertainty:

            # A solução ótima está dentro do intervalo [ak,bk].

            # Escolhendo o ponto do meio como solução ótima:

            search_magnitude=(bk+ak)/2
            termination=False

        else:
            k+=1
    return search_magnitude
```

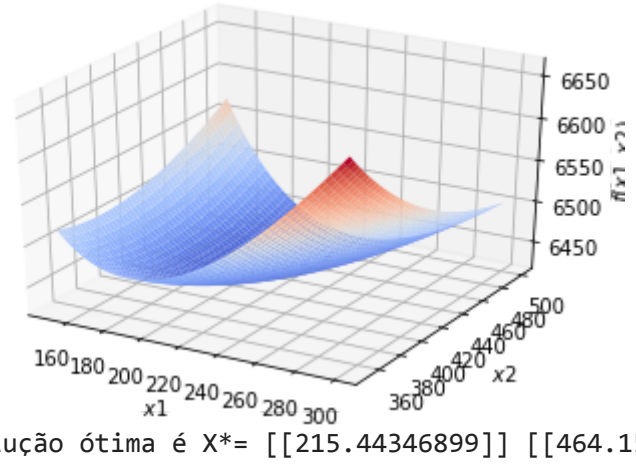
#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══

```
# Definindo o método do gradiente ("steepest descent")

# Definindo um dataframe (pandas.dataframe) para salvar as soluções e direções para cada iteração:

points_checked=pd.DataFrame(columns=["xk","dk","alfa","x(k+1)","gradiente","fun obj"])

# Escolhendo o ponto inicial, tamanho da incerteza e calculando a primeira direção do gradiente:
len_uncert=1e-9
Xk=np.matrix([[100],[100]])
n=int(Xk.shape[0])
dk=-gradf(Xk[0,0],Xk[1,0])
termination=True

while termination:

    # Calculando a magnitude da direção (alfad) usando o método da seção áurea ("golden search"):

    alfad=golden_search(0,100,1e-9,Xk,dk)

    # Calculando o novo ponto:

    Xk1=Xk+alfad*dk

    # Salvando a informação no dataframe:

    points_checked=points_checked.append({"xk":(np.round(Xk[0,0],2),np.round(Xk[1,0],2)),
                                          "dk":(np.round(dk[0,0],2),np.round(dk[1,0],2)),
                                          "alfa":np.round(alfad,3),
                                          "x(k+1)":np.round(Xk1,2),
                                          "gradiente":np.round(gradf(Xk1[0,0],Xk1[1,0]),2),
                                          "fun obj":np.round(fo(Xk1[0,0],Xk1[1,0]),2)}, ignore_index=True)

    # Atualizando a direção na direção do gradiente "steepest":

    dk=-gradf(Xk1[0,0],Xk1[1,0])

    # Checando o critério de término:

    if LA.norm(gradf(Xk1[0,0],Xk1[1,0]))<len_uncert:
        termination=False
        print("A solução ótima é X*=",Xk1[0],Xk1[1], "e a função objetivo no ótimo é = ",np.round(fo(Xk1[0,0],Xk1[1,0]),2)  )

    #Atualizando Xk:

    Xk=Xk1
```

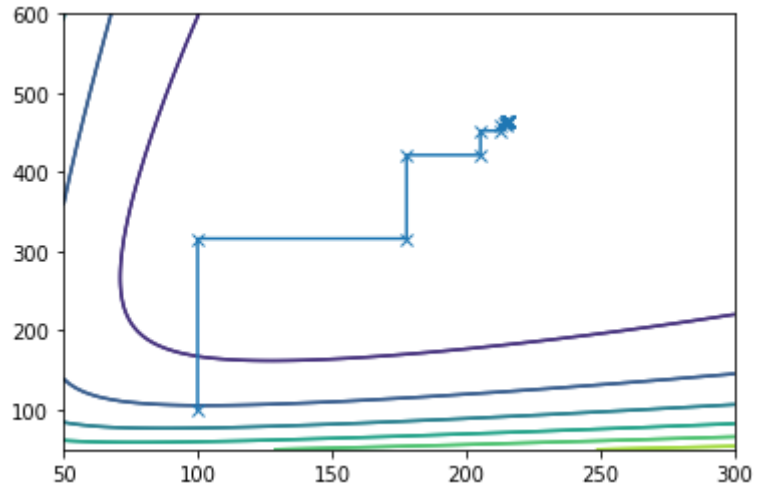
#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══

```
print('Nº de steps que o método Steepest Descent levou para convergir: ', len(points_checked))

print("Nº de avaliações da função objetivo = " + str(count))

print("Tabela com as iterações: ")

print(points_checked)

print("Caminho do método Steepest Descent sobre as curvas de níveis de f:")
contour = contour(f)

y = points_checked['xk'].values.astype(dtype=[('f0', '<f8'), ('f1', '<f8')])      # Essa é uma transformação que será explicada
                                                                                   # no arquivo EXTRA
```

```
z = y.view(np.float64).reshape(y.shape + (-1,))

Contour_path = contour_travel(z, f)
```




```
A solução ótima é X*= [[215.44346899]] [[464.15888336]] e a função objetivo no ótimo é =  6421.49
Nº de steps que o método Steepest Descent levou para convergir:  275
Nº de avaliações da função objetivo = 15402
Tabela com as iterações:
                   xk             dk  ...          gradiente  fun obj
0          (100, 100)  (-0.0, 23.21)  ...   [[-9.71], [0.0]]  6800.34
1     (100.0, 316.23)   (9.71, -0.0)  ...   [[0.0], [-1.66]]  6521.37
2    (177.83, 316.23)   (-0.0, 1.66)  ...  [[-1.42], [-0.0]]  6445.77
3     (177.83, 421.7)    (1.42, 0.0)  ...    [[0.0], [-0.3]]  6427.64
4     (205.35, 421.7)    (-0.0, 0.3)  ...  [[-0.3], [-0.01]]  6423.02
..                ...            ...  ...                ...      ...
270  (215.44, 464.16)   (-0.0, 0.0)  ...    [[-0.0], [0.0]]  6421.49
271  (215.44, 464.16)   (0.0, -0.0)  ...    [[0.0], [-0.0]]  6421.49
272  (215.44, 464.16)   (-0.0, 0.0)  ...    [[-0.0], [0.0]]  6421.49
273  (215.44, 464.16)   (0.0, -0.0)  ...    [[0.0], [-0.0]]  6421.49
274  (215.44, 464.16)   (-0.0, 0.0)  ...    [[-0.0], [0.0]]  6421.49

[275 rows x 6 columns]
Caminho do método Steepest Descent sobre as curvas de níveis de f:
```



2º Método de otimização indireto - ***Gradiente Conjugado ("Conjugate Gradient: Fletcher-Reeves")***:

```
# Inicializando o contador externamente
count = 0

# Definindo o contador que entrará junto com a função fun(x), que dirá quantas vezes fun(x) foi avaliada

def incCount():
    global count
    count = count+1

# Definindo a função objetivo fo(x1,x2):

def fo(x1,x2):
    incCount()
    return 2342.1971697181*x1**0.2857 + 8730.1851*(x2/x1)**0.2857 + 62823.6915040394*(1/x2)**0.2857 - 26190.5553


# Definindo o gradiente da função:

def gradf(x1,x2):
    return np.matrix([[669.165731388461*x1**(-0.7143) - 2494.21388307*(x2/x1)**0.2857/x1],
                      [2494.21388307*(x2/x1)**0.2857/x2 - 17948.7286627041*(1/x2)**0.2857/x2]])

# Visualização gráfica da função para certificar que está correta:

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
```

```
fig = plt.figure()
ax = fig.gca(projection='3d')

X = np.linspace(150, 300, 50)
Y = np.linspace(350, 500, 50)
X, Y = np.meshgrid(X, Y)
Z = fo(X,Y)

surf = ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$f(x1,x2)$');

plt.show()
```

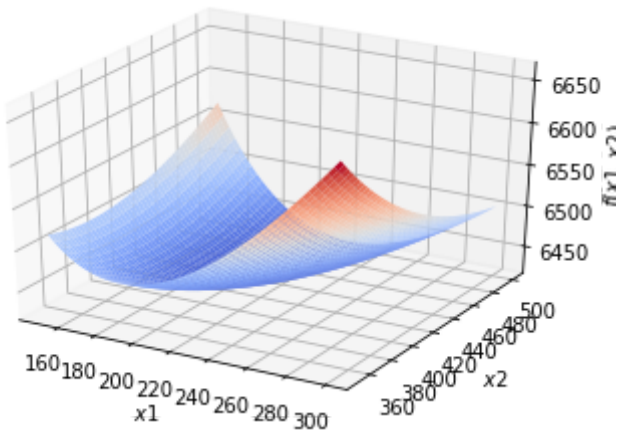
```
#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══
```

```
# Definindo o método do gradiente conjugado ("Fletcher Reeves")

# Definindo um dataframe (pandas.dataframe) para salvar as soluções e direções para cada iteração:

points_checked=pd.DataFrame(columns=["xk","alfacon","dk","alfa","x(k+1)","gradiente","fun obj"])

# Escolhendo o ponto inicial, tamanho da incerteza e calculando a primeira direção do gradiente:
len_uncert=1e-9
Xk=np.matrix([[100],[100]])
n=int(Xk.shape[0])
dk=-gradf(Xk[0,0],Xk[1,0])
termination=True

while termination:

    # Calculando a magnitude da direção (alfad) usando o método da seção áurea ("golden search"):

    alfad=golden_search(0,100,1e-9,Xk,dk)

    # Calculando o novo ponto:

    Xk1=Xk+alfad*dk

    # Calculando "alfacon" para obter uma direção conjugada:

    alfacon=(LA.norm(gradf(Xk1[0,0],Xk1[1,0]))**2)/(LA.norm(gradf(Xk[0,0],Xk[1,0]))**2)

    # Salvando a informação no dataframe:

    points_checked=points_checked.append({"xk":(np.round(Xk[0,0],2),np.round(Xk[1,0],2)),
                                          "alfacon":np.round(alfacon,3),
                                          "dk":(np.round(dk[0,0],2),np.round(dk[1,0],2)),
                                          "alfa":np.round(alfad,3),
                                          "x(k+1)":np.round(Xk1,2),
                                          "gradiente":np.round(gradf(Xk1[0,0],Xk1[1,0]),2),
                                          "fun obj":np.round(fo(Xk1[0,0],Xk1[1,0]),2)}, ignore_index=True)

    # Atualizando a direção gradiente:

    dk=-gradf(Xk1[0,0],Xk1[1,0])+alfacon*dk

    # Checando o critério de término:

    if LA.norm(gradf(Xk1[0,0],Xk1[1,0]))<len_uncert:
        termination=False
        print("A solução ótima é X*=",Xk1[0],Xk1[1], "e a função objetivo no ótimo é = ",np.round(fo(Xk1[0,0],Xk1[1,0]),2)  )

    #Atualizando Xk:

    Xk=Xk1
```

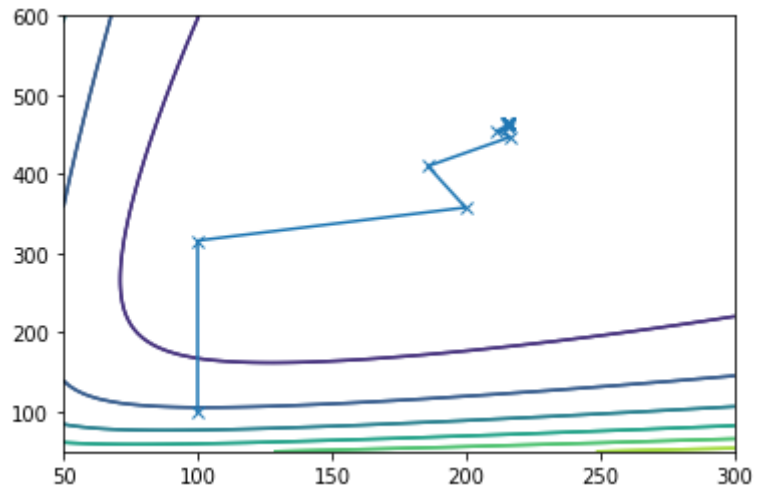
```
#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══
```

```
print('Nº de steps que o método Gradiente Conjugado levou para convergir: ', len(points_checked))

print("Nº de avaliações da função objetivo = " + str(count))

print("Tabela com as iterações: ")

print(points_checked)
```

```
print("Caminho do método Gradiente Conjugado sobre as curvas de níveis de f:")
contour = contour(f)

y = points_checked['xk'].values.astype(dtype=[('f0', '<f8'), ('f1', '<f8')])

z = y.view(np.float64).reshape(y.shape + (-1,))

Contour_path = contour_travel(z, f)
```




```
A solução ótima é X*= [[215.44346902]] [[464.15888341]] e a função objetivo no ótimo é =  6421.49
Nº de steps que o método Gradiente Conjugado levou para convergir:  41
Nº de avaliações da função objetivo = 2298
Tabela com as iterações:
                  xk  alfacon  ...           gradiente  fun obj
0         (100, 100)    0.175  ...    [[-9.71], [0.0]]  6800.34
1    (100.0, 316.23)    0.015  ...   [[0.47], [-1.11]]  6469.48
2   (199.79, 357.99)    0.486  ...  [[-0.81], [-0.22]]  6438.23
3   (185.75, 410.23)    0.087  ...   [[0.19], [-0.16]]  6422.92
4    (216.4, 446.94)    0.107  ...  [[-0.07], [-0.04]]  6421.81
5   (211.58, 455.18)    0.495  ...   [[0.04], [-0.04]]  6421.57
6   (215.58, 459.87)    0.063  ...   [[-0.01], [-0.0]]  6421.50
7   (214.84, 463.13)    0.181  ...    [[0.0], [-0.01]]  6421.49
8   (215.35, 463.42)    0.391  ...    [[-0.0], [-0.0]]  6421.49
9    (215.3, 463.95)    0.053  ...     [[0.0], [-0.0]]  6421.49
10  (215.44, 464.07)    0.296  ...    [[-0.0], [-0.0]]  6421.49
11  (215.42, 464.11)    0.246  ...     [[0.0], [-0.0]]  6421.49
12  (215.45, 464.14)    0.058  ...    [[-0.0], [-0.0]]  6421.49
13  (215.44, 464.15)    0.436  ...     [[0.0], [-0.0]]  6421.49
14  (215.44, 464.16)    0.139  ...    [[-0.0], [-0.0]]  6421.49
15  (215.44, 464.16)    0.075  ...     [[0.0], [-0.0]]  6421.49
16  (215.44, 464.16)    0.594  ...    [[-0.0], [-0.0]]  6421.49
17  (215.44, 464.16)    0.034  ...     [[0.0], [-0.0]]  6421.49
18  (215.44, 464.16)    0.936  ...    [[-0.0], [-0.0]]  6421.49
19  (215.44, 464.16)    0.375  ...     [[-0.0], [0.0]]  6421.49
20  (215.44, 464.16)    4.024  ...     [[0.0], [-0.0]]  6421.49
21  (215.44, 464.16)    2.174  ...     [[0.0], [-0.0]]  6421.49
22  (215.44, 464.16)    1.662  ...     [[0.0], [-0.0]]  6421.49
23  (215.44, 464.16)    1.136  ...     [[0.0], [-0.0]]  6421.49
24  (215.44, 464.16)    0.811  ...      [[0.0], [0.0]]  6421.49
25  (215.44, 464.16)    0.223  ...      [[0.0], [0.0]]  6421.49
26  (215.44, 464.16)    1.441  ...     [[-0.0], [0.0]]  6421.49
27  (215.44, 464.16)    2.500  ...     [[-0.0], [0.0]]  6421.49
28  (215.44, 464.16)    0.382  ...     [[-0.0], [0.0]]  6421.49
29  (215.44, 464.16)    0.674  ...     [[0.0], [-0.0]]  6421.49
30  (215.44, 464.16)    1.180  ...     [[0.0], [-0.0]]  6421.49
31  (215.44, 464.16)    0.446  ...    [[-0.0], [-0.0]]  6421.49
32  (215.44, 464.16)    0.388  ...    [[-0.0], [-0.0]]  6421.49
33  (215.44, 464.16)    0.731  ...     [[0.0], [-0.0]]  6421.49
34  (215.44, 464.16)    0.125  ...      [[0.0], [0.0]]  6421.49
35  (215.44, 464.16)    0.617  ...      [[0.0], [0.0]]  6421.49
36  (215.44, 464.16)    0.265  ...      [[0.0], [0.0]]  6421.49
37  (215.44, 464.16)    0.057  ...      [[0.0], [0.0]]  6421.49
38  (215.44, 464.16)    0.432  ...      [[0.0], [0.0]]  6421.49
39  (215.44, 464.16)    0.095  ...      [[0.0], [0.0]]  6421.49
40  (215.44, 464.16)    0.046  ...      [[0.0], [0.0]]  6421.49

[41 rows x 7 columns]
Caminho do método Gradiente Conjugado sobre as curvas de níveis de f:
```



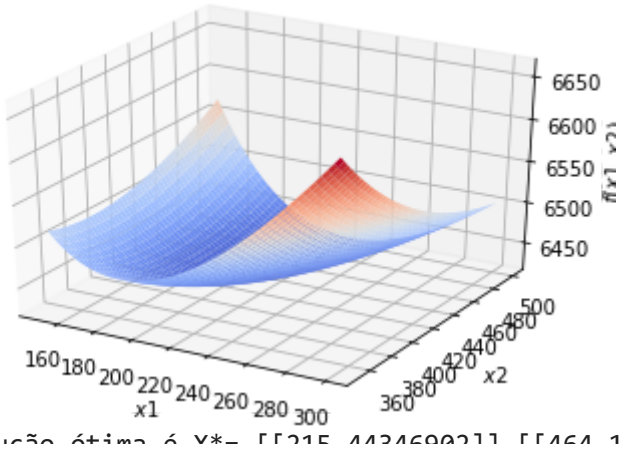
3º Método de otimização indireto - ***BFGS (Broyden-Fletcher-Goldfarb-Shanno)***:

```
# Inicializando o contador externamente

count = 0

# Definindo o contador que entrará junto com a função fun(x), que dirá quantas vezes fun(x) foi avaliada

def incCount():
    global count
    count = count+1

# Definindo a função objetivo fo(x1,x2):

def fo(x1,x2):
    incCount()
    return 2342.1971697181*x1**0.2857 + 8730.1851*(x2/x1)**0.2857 + 62823.6915040394*(1/x2)**0.2857 - 26190.5553


# Definindo o gradiente da função:

def gradf(x1,x2):
    return np.matrix([[669.165731388461*x1**(-0.7143) - 2494.21388307*(x2/x1)**0.2857/x1],
                      [2494.21388307*(x2/x1)**0.2857/x2 - 17948.7286627041*(1/x2)**0.2857/x2]])

# Visualização gráfica da função para certificar que está correta:

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

fig = plt.figure()
ax = fig.gca(projection='3d')

X = np.linspace(150, 300, 50)
Y = np.linspace(350, 500, 50)
X, Y = np.meshgrid(X, Y)
Z = fo(X,Y)

surf = ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$f(x1,x2)$');

plt.show()

#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══

# Definindo o método do BFGS (Broyden-Fletcher-Goldfarb-Shanno)

# Definindo um dataframe (pandas.dataframe) para salvar as soluções e direções para cada iteração:

points_checked=pd.DataFrame(columns=["xk","Dk","dk","alfa","x(k+1)","gradiente","fun obj"])

# Escolhendo o ponto inicial, tamanho da incerteza e calculando a primeira direção do gradiente:

len_uncert=1e-9
Xk=np.matrix([[100],[100]])
n=int(Xk.shape[0])
Dk=np.eye(n)               #Dk = matriz "H(xk)" [inicialmente matriz identidade]
termination=True

while termination:

    #Calculando as direções usando a matriz Dk

    dk=-Dk*gradf(Xk[0,0],Xk[1,0])

    # Calculando a magnitude da direção (alfad) usando o método da seção áurea ("golden search"):

    alfad=golden_search(0,100,1e-9,Xk,dk)

    # Calculando o novo ponto:

    Xk1=Xk+alfad*dk

    # Calculando Ck usando o método BFGS:                        # Ck são aqueles 2 termos de correção de H(xk)
```

```
pk=Xk1-Xk                                                    # pk = "delta x" (Ver notação do slide 57)

qk=gradf(Xk1[0,0],Xk1[1,0])-gradf(Xk[0,0],Xk[1,0])           # qk = "delta f"

Ck=(((pk*pk.T)/(pk.T*qk))*float((1+((qk.T*Dk*qk)/(pk.T*qk)))))-((Dk*qk*pk.T+pk*qk.T*Dk)/(pk.T*qk))

    # Salvando a informação no dataframe:

points_checked=points_checked.append({"xk":(np.round(Xk[0,0],2),np.round(Xk[1,0],2)),
                                      "Dk":np.round(Dk,3),
                                      "dk":(np.round(dk[0,0],2),np.round(dk[1,0],2)),
                                      "alfa":np.round(alfad,3),
                                      "x(k+1)":np.round(Xk1,2),
                                      "gradiente":np.round(gradf(Xk1[0,0],Xk1[1,0]),2),
                                      "fun obj":np.round(fo(Xk1[0,0],Xk1[1,0]),2)}, ignore_index=True)

# Atualizando Dk

Dk=Dk+Ck

# Checando o critério de término:

if LA.norm(gradf(Xk1[0,0],Xk1[1,0]))<len_uncert:
    termination=False
    print("A solução ótima é X*=",Xk1[0],Xk1[1], "e a função objetivo no ótimo é = ",np.round(fo(Xk1[0,0],Xk1[1,0]),2)  )
    #Atualizando Xk:

    Xk=Xk1
```

```
#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══
```

```
print('Nº de steps que o método BFGS levou para convergir: ', len(points_checked))

print("Nº de avaliações da função objetivo = " + str(count))

print("Tabela com as iterações: ")

print(points_checked)

print("Caminho do método BFGS sobre as curvas de níveis de f:")
contour = contour(f)

y = points_checked['xk'].values.astype(dtype=[('f0', '<f8'), ('f1', '<f8')])

z = y.view(np.float64).reshape(y.shape + (-1,))

Contour_path = contour_travel(z, f)
```



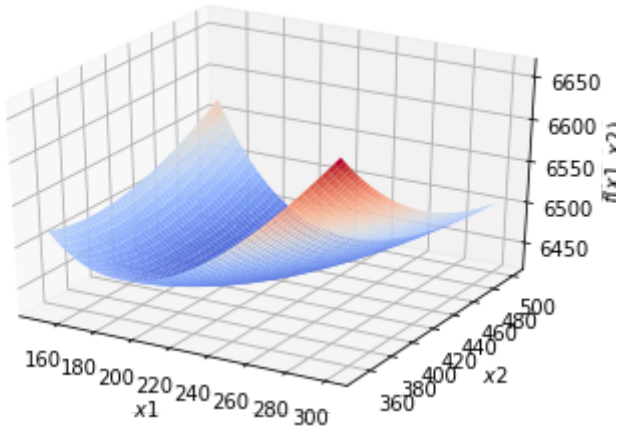
A solução ótima é X*= [[215.443690211] [[464.158883371] e a função objetivo no ótimo é = -6421.49

4º Método de otimização indireto - ***DFP (Davidon-Fletcher-Powell)***:

Tabela com as iteracões:

```
# Inicializando o contador externamente

count = 0

# Definindo o contador que entrará junto com a função fun(x), que dirá quantas vezes fun(x) foi avaliada

def incCount():
    global count
    count = count+1

# Definindo a função objetivo fo(x1,x2):

def fo(x1,x2):
    incCount()
    return 2342.1971697181*x1**0.2857 + 8730.1851*(x2/x1)**0.2857 + 62823.6915040394*(1/x2)**0.2857 - 26190.5553


# Definindo o gradiente da função:

def gradf(x1,x2):
    return np.matrix([[669.165731388461*x1**(-0.7143) - 2494.21388307*(x2/x1)**0.2857/x1],
                      [2494.21388307*(x2/x1)**0.2857/x2 - 17948.7286627041*(1/x2)**0.2857/x2]])

# Visualização gráfica da função para certificar que está correta:

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

fig = plt.figure()
ax = fig.gca(projection='3d')

X = np.linspace(150, 300, 50)
Y = np.linspace(350, 500, 50)
X, Y = np.meshgrid(X, Y)
Z = fo(X,Y)

surf = ax.plot_surface(X, Y, Z,cmap=cm.coolwarm)
ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$f(x1,x2)$');

plt.show()

#=========╬=========╬=========╬=========╬=========╬=========╬=========╬=========╬=========╬=========╬=========╬=========╬=========╬=========╬=========╬=

# Definindo o método do DFP (Davidon-Fletcher-Powell)

# Definindo um dataframe (pandas.dataframe) para salvar as soluções e direções para cada iteração:

points_checked=pd.DataFrame(columns=["xk","Dk","dk","alfa","x(k+1)","gradiente","fun obj"])

# Escolhendo o ponto inicial, tamanho da incerteza e calculando a primeira direção do gradiente:

len_uncert=1e-9
Xk=np.matrix([[100],[100]])
n=int(Xk.shape[0])
Dk=np.eye(n)               #Dk = matriz "W(xk)" [inicialmente matriz identidade]
termination=True

while termination:

    #Calculando as direções usando a matriz Dk
```

```
dk=-Dk*gradf(Xk[0,0],Xk[1,0])

# Calculando a magnitude da direção (alfad) usando o método da seção áurea ("golden search"):

alfad=golden_search(0,100,1e-9,Xk,dk)

# Calculando o novo ponto:

Xk1=Xk+alfad*dk

#Calculando Ck usando o método DFP:                                  # Ck são aqueles 2 termos de correção de W(xk)

pk=Xk1-Xk                                                           # pk = delta x (Ver notação do slide 58)
qk=gradf(Xk1[0,0],Xk1[1,0])-gradf(Xk[0,0],Xk[1,0])                  # qk = delta f
Ck=((pk*pk.T)/(pk.T*qk))-((Dk*qk*qk.T*Dk)/(qk.T*Dk*qk))

    # Salvando a informação no dataframe:

points_checked=points_checked.append({"xk":(np.round(Xk[0,0],2),np.round(Xk[1,0],2)),
                                      "Dk":np.round(Dk,3),
                                      "dk":(np.round(dk[0,0],2),np.round(dk[1,0],2)),
                                      "alfa":np.round(alfad,3),
                                      "x(k+1)":np.round(Xk1,2),
                                      "gradiente":np.round(gradf(Xk1[0,0],Xk1[1,0]),2),
                                      "fun obj":np.round(fo(Xk1[0,0],Xk1[1,0]),2)}, ignore_index=True)

    # Atualizando Dk

    Dk=Dk+Ck

    # Checando o critério de término:

    if LA.norm(gradf(Xk1[0,0],Xk1[1,0]))<len_uncert:
        termination=False
        print("A solução ótima é X*=",Xk1[0],Xk1[1], "e a função objetivo no ótimo é = ",np.round(fo(Xk1[0,0],Xk1[1,0]),2)  )

    #Atualizando Xk:

    Xk=Xk1
#═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══════╬═══

print('Nº de steps que o método DFP levou para convergir: ', len(points_checked))

print("Nº de avaliações da função objetivo = " + str(count))

print("Tabela com as iterações: ")

print(points_checked)

print("Caminho do método DFP sobre as curvas de níveis de f:")
contour = contour(f)
y = points_checked['xk'].values.astype(dtype=[('f0', '<f8'), ('f1', '<f8')])

z = y.view(np.float64).reshape(y.shape + (-1,))

Contour_path = contour_travel(z, f)
```




```
A solução ótima é X*= [[215.443469]] [[464.15888336]] e a função objetivo no ótimo é =  6421.49
Nº de steps que o método DFP levou para convergir:  6680
Nº de avaliações da função objetivo = 374082
Tabela com as iterações:
                    xk  ...  fun obj
0           (100, 100)  ...  6800.34
1      (100.0, 316.23)  ...  6469.48
2     (199.79, 357.99)  ...  6421.78
3     (211.09, 461.35)  ...  6421.49
4     (215.38, 463.67)  ...  6421.49
...                ...  ...      ...
6675  (215.44, 464.16)  ...  6421.49
6676  (215.44, 464.16)  ...  6421.49
```

ADICIONAL: Método de otimização indireto - ***BFGS (Broyden-Fletcher-Goldfarb-Shanno)*** por meio do Scipy.optimize.minimize:

```
import numpy as np
from scipy.optimize import minimize

x0 = np.array([100, 100])

# Definindo a função objetivo f(x):

def f(x):
    return 2342.1971697181*x[0]**0.2857 + 8730.1851*(x[1]/x[0])**0.2857 + 62823.6915040394*(1/x[1])**0.2857 - 26190.5553


# Definindo o gradiente da função:

def gradf(x):
    return np.array([669.165731388461*x[0]**(-0.7143) - 2494.21388307*(x[1]/x[0])**0.2857/x[0],
                     2494.21388307*(x[1]/x[0])**0.2857/x[1] - 17948.7286627041*(1/x[1])**0.2857/x[1]])

res = minimize(f, x0, method='BFGS', jac=gradf, options={'gtol': 1e-9, 'disp': True})

res.x

     Optimization terminated successfully.
              Current function value: 6421.490224
              Iterations: 18
              Function evaluations: 21
              Gradient evaluations: 21
     array([215.443469  , 464.15888336])
```

Resposta: (b) Podemos fazer comparações entre os métodos, de acordo com a seguinte tabela:

Métodos	Ponto ótimo: x(p2,p3)	Nº de iterações	Nº de avaliações da função objetivo	Função objetivo no ótimo
Rosenbrock	(215,45; 464,15)	8	372	6421,49
Powell (scipy.minimize)	(215,44; 464,16)	5	175	6421,49
Gradiente	(215,44; 464,16)	275	15402	6421,49
Gradiente conjugado	(215,44; 464,16)	41	2298	6421,49
DFP	(215,44; 464,16)	16966	950098	6421,49
BFGS	(215,44; 464,16)	6680	374082	6421,49
BFGS (scipy.minimize)	(215,44; 464,16)	18	21	6421,49

Como podemos verificar, praticamente todos os métodos encontraram o mesmo ponto ótimo, com a mesmo valor da função objetivo. O único método que tem uma ligeira alteração no ponto ótimo é o de Rosenbrock (~0,01 em p2, p3).

A convergência dos métodos foi parecida, porque a função objetivo no caso é bem definida, diferenciável em todo o intervalo de x, e convexa. Se as derivadas de f(x) não estivessem disponíveis, não poderíamos usar os métodos indiretos (em laranja).

A maior diferença que podemos detectar para este caso, é a tendência de um maior número de iterações/avaliações da função objetivo para os métodos indiretos; especialmente para os métodos do DFP e BFGS implementados.

9) Resolva novamente os exercícios (4b) e (4d) utilizando o método SQP e levando-se em conta as restrições do problema e $L(x) \geq 0$.

Solução:

Quando resolvemos os problemas $4(b)$ e $4(d)$, utilizamos o método de otimização ***SLSQP***. Na realidade, o SLSQP é análogo ao ***SQP***.

Mais precisamente, SQP e SLSQP apresentam o mesmo subproblema da Programação Quadrática (***QP***) em cada etapa do algoritmo.

No SQP, o problema de QP é resolvido por métodos de Programação Quadrática.

Em SLSQP, para resolver o problema de QP, você deve fatorar em LDL^{-1} o lagrangeano da Hessiana e então resolver um problema de mínimos quadrados lineares:

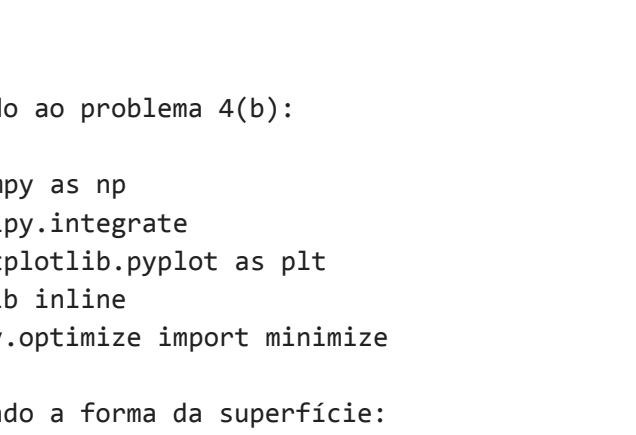
$$
\begin{aligned}
&\underset{d\in\mathbb{R}^n}{min} \left\| \left(D^k\right)^{1/2}\left(L^k\right)^T d + \left(D^k\right)^{-1/2}\left(L^k\right)^{-1}\triangledown f(x^k) \right\| \\
&\triangledown g_j(x^k)d + g_j(x^k) = 0, \qquad j = 1,\dots,m_c, \\
&\triangledown g_j(x^k)d + g_j(x^k) \geq 0, \qquad j = m_c+1,\dots,m
\end{aligned}
$$

Como os dois métodos resolvem o mesmo problema QP, vamos continuar considerando o SLSQP aqui.

E, retornando aos problemas $4(b)$ e $4(d)$, já havíamos considerado as restrições dadas pelo problema original.

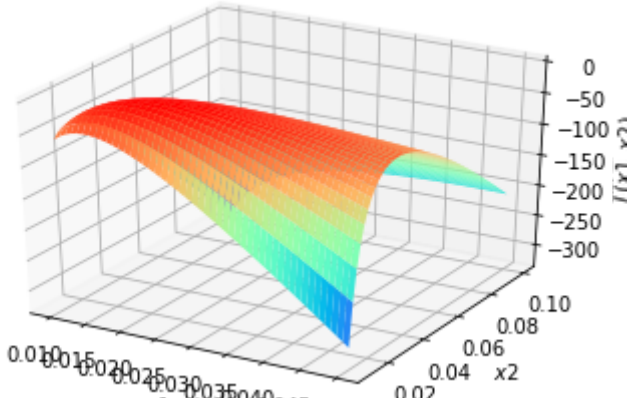
A única restrição que está faltando então é $L(x) \geq 0$:

```
#Retornando ao problema 4(b):

import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.optimize import minimize

#Verificando a forma da superfície:

from matplotlib import cm

x1 = np.linspace(1E-3, 2E-2, 50)
x2 = np.linspace(1E-3, 2E-2, 50)
X, Y = np.meshgrid(x1, x2)

Z = 129.93 - 0.5/X - 4000*Y - 25*(X/Y)

fig, ax = plt.subplots(subplot_kw={'projection': '3d'})
ax.plot_surface(X, Y, Z, cmap=cm.rainbow)

ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$L(x1,x2)$');
```




```
def func(x, a=129.93, b=0.5, c=4000, d=25, sign=-1.0):
    return sign*(a - b/x[0] - c*x[1] - d*x[0]/x[1])

def func_deriv(x, a=129.93, b=0.5, c=4000, d=25, sign=-1.0):
    dfdx0 = sign*(0 + b/(x[0])**2 + 0 - d/x[1])
    dfdx1 = sign*(0 + 0 - c + d*x[0]/(x[1])**2)
    return np.array([ dfdx0, dfdx1 ])

from scipy.optimize import Bounds
```

```
from scipy.optimize import bounds
bounds = Bounds([1E-9, 1E-9], [1.0, 1.0])

#Agora, além das restrições de desigualdade x1 <= 0,02, x2 <= x1;
#temos que adicionar também a desigualdade func(x) >= 0:

ineq_cons = {'type': 'ineq',
             'fun' : lambda x: np.array([0.02 - x[0],
                                         x[0] - x[1],
                                         func(x, sign=1.0)]),
             'jac' : lambda x: np.array([[-1.0, 0.0],
                                         [1.0, -1.0],
                                         func_deriv(x, sign=1.0)])}


#Executando a otimização com restrições (constraints = ineq_cons):

x0 = np.array([0.5, 0.5])
res = minimize(func, x0, jac=func_deriv, constraints=ineq_cons,
               method='SLSQP', options={'ftol': 1e-9, 'disp': True},
               bounds = bounds)
print(res.x)

     Optimization terminated successfully.    (Exit mode 0)
                Current function value: -19.409055040739943
                Iterations: 14
                Function evaluations: 21
                Gradient evaluations: 14
     [0.01357208 0.00921008]


#Resultado em notação científica:

scientific_notation1 = "{:.2e}".format(res.x[0])

scientific_notation2 = "{:.2e}".format(res.x[1])

print(scientific_notation1 + "," + scientific_notation2)

     1.36e-02,9.21e-03


#Valor da função objetivo (lucro) no ponto máximo:

#Lembrar de alterar o sinal novamente para sign=1.0:

def func(x, a=129.93, b=0.5, c=4000, d=25, sign=1.0):
    return sign*(a - b/x[0] - c*x[1] - d*x[0]/x[1])

result = func([1.36E-2, 9.21E-3])

print(result)

     19.40889889506292


#Retornando ao problema 4(d):

import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.optimize import minimize

#Verificando a forma da superfície:

from matplotlib import cm

x1 = np.linspace(1E-2, 5E-2, 50)
x2 = np.linspace(1E-2, 1E-1, 50)
X, Y = np.meshgrid(x1, x2)

Z = 279.72 - 2.0/X - 4000*Y - 100*(X/Y)

fig, ax = plt.subplots(subplot_kw={'projection': '3d'})
ax.plot_surface(X, Y, Z, cmap=cm.rainbow)

ax.set_xlabel('$x1$')
ax.set_ylabel('$x2$')
ax.set_zlabel('$L(x1,x2)$');
```




```
def func(x, a=279.72, b=2.0, c=4000, d=100, sign=-1.0):
    return sign*(a - b/x[0] - c*x[1] - d*x[0]/x[1])

def func_deriv(x, a=279.72, b=2.0, c=4000, d=100, sign=-1.0):
    dfdx0 = sign*(0 + b/(x[0])**2 + 0 - d/x[1])
    dfdx1 = sign*(0 + 0 - c + d*x[0]/(x[1])**2)
    return np.array([ dfdx0, dfdx1 ])

from scipy.optimize import Bounds
bounds = Bounds([1E-9, 1E-9], [1.0, 1.0])

#Agora, além das restrições de desigualdade x1 <= 0,02, x2 <= x1;
#temos que adicionar também a desigualdade func(x) >= 0:
ineq_cons = {'type': 'ineq',
             'fun' : lambda x: np.array([0.02 - x[0],
                                         x[0] - x[1],
                                         func(x, sign=1.0)]),
             'jac' : lambda x: np.array([[-1.0, 0.0],
                                         [1.0, -1.0],
                                         func_deriv(x, sign=1.0)])}


#E a otimização com restrições (OPCIONAL - constraints = ineq_cons):

x0 = np.array([0.5, 0.5])
res = minimize(func, x0, jac=func_deriv, constraints=ineq_cons,
               method='SLSQP', options={'ftol': 1e-9, 'disp': True},
               bounds = bounds)
print(res.x)

     Inequality constraints incompatible    (Exit mode 4)
                 Current function value: 0.2642368615241253
                 Iterations: 11
                 Function evaluations: 33
                 Gradient evaluations: 10
     [0.02001582 0.02001583]


#Valor da função objetivo (lucro) no ponto máximo (com restrições):

#Lembrar de alterar o sinal novamente para sign=1.0:
def func(x, a=279.72, b=2.0, c=4000, d=100, sign=1.0):
    return sign*(a - b/x[0] - c*x[1] - d*x[0]/x[1])

result = func([0.02001582, 0.02001583])

print(result)

     -0.26423255819140934
```

Resposta: (9) A nova restrição $L(x) \geq 0$ efetivamente não teve nenhum impacto sobre a solução do problema ***4(b)***. O novo ponto ótimo e o valor da função objetivo (Lucro) permaneceram inalterados:

> $x = (1,36 \cdot 10^{-2}, 9,21 \cdot 10^{-3})$
> $L(x) = 19,41$

Obviamente, não esperava-se um resultado diferente, visto que a solução do problema 4(b) já satisfazia a nova restrição imposta $L(x) \geq 0$.

Já para a solução do problema ***4(d)***, essa nova restrição proporcionou uma leve alteração no resultado, praticamente imperceptível:

> $x = (2,002 \cdot 10^{-2}, 2,002 \cdot 10^{-2})$

> $$L(x) = -0,26$$

A solução que tínhamos obtido no passado, no problema 4(d), era:

> $$x = (2,00 \cdot 10^{-2}, 2,00 \cdot 10^{-2})$$
> $$L(x) = -0,28$$

Esse não era o resultado que se esperava obter, já que essa nova condição imposta gerou uma solução que não satisfaz 2 restrições: $x_1 \leq 0$, e a própria restrição $L(x) \geq 0$.

Sendo mais criterioso, ela também viola a terceira condição: $x_2 \leq x_1$, na última casa decimal: $x = [0.02001582, 0.02001583]$.

Ou seja, essa nova restrição fez com que todas as outras também fossem violadas.

Esse é um resultado estranho, pois esperava-se simplesmente que o algoritmo não convergisse, já que não existe nenhuma solução possível que atenda às 3 restrições impostas. No entanto, ele convergiu para uma solução que viola as 3 condições ao mesmo tempo.

O algoritmo até indica que reconheceu problemas nesse caso, com a mensagem no início de seu output:

> *Inequality constraints incompatible (Exit mode 4)*

Mas mesmo assim, ele ainda fornece uma resposta final. Provavelmente porque deve existir alguma função dentro do próprio algoritmo do "minimize" que converge para a solução mais próxima desconsiderando-se essa última restrição imposta.